"Express Mail" mailing label number:

EL989617082US

METHOD AND APPARATUS FOR IMPLEMENTING PROCESSOR INSTRUCTIONS FOR ACCELERATING PUBLIC-KEY CRYPTOGRAPHY

Sheueling Chang Shantz
Hans Eberle
Nils Gura
Lawrence Spracklen
Leonard Rarick

CROSS-REFERENCE TO RELATED APPLICATION(S)

[1001] This application claims benefit under 35 U.S.C. § 119(e) of application no. 60/_________, filed June 30, 2003, ENTITLED "Accelerating Public-Key Cryptography", naming Sheueling Chang Shantz et al. as inventors, attorney docket number 004-9388-V, which application is incorporated herein by reference.

BACKGROUND

Field of the Invention

[1002] This invention relates to computer systems and more particularly to cryptographic computations performed therein.

Description of the Related Art

[1003] Internet standards such as Secure Socket Layer (SSL) and IP security (IPsec) rely on public-key cryptosystems for scalable key management. With the enormous growth of the World-Wide-Web and, in particular, the ever increasing deployment of e-commerce applications based on https (http over SSL), it has become important to efficiently support cryptographic computations in computer systems, particularly server systems.

[1004] Public-key cryptosystems such as the Rivest-Shamir-Adleman (RSA) public-key algorithm and the Diffie-Hellman (DH) key exchange scheme require modular exponentiation with operands of at least 512 bits. Modular exponentiation is computed using a series of modular multiplications and squarings. A newly standardized public-key system, the Elliptic Curve Cryptography (ECC), also uses large integer arithmetic, even though it requires much smaller key sizes. The Elliptic Curve public-key cryptographic systems operate in both integer and binary polynomial fields. A typical RSA operation requires a 1024-bit modular exponentiation (or two 512-bit modular exponentiations using the Chinese Remainder Theorem). RSA key sizes are expected to grow to 2048 bits in the near future. A 1024-bit modular exponentiation includes a sequence of large integer modular multiplications; each, in turn, is further broken up into many word-size multiplications. In total, a 1024-bit modular exponentiation requires over 1.6 million 64-bit multiplications. Thus, public-key algorithms are compute-intensive with relatively few data movements. The computations required are generic arithmetic functions such as integer multiplications and additions. Given those characteristics, public-key algorithms can be well supported by general-purpose processors.

[1005] In order to better support cryptography applications, it would be desirable to enhance the capability of general-purpose processors to accelerate public-key computations.

SUMMARY

[1006] In one embodiment, the invention provides, a method for operating a processor that includes, in response to executing a single arithmetic instruction, multiplying a first number by a second number, and adding implicitly a partial result from a previously executed single arithmetic instruction to generate a result that represents the first number multiplied by the second number summed with the partial result. The method may further include storing a high order portion of the generated result in an extended carry register as a next partial result for use with execution of a subsequent single arithmetic instruction.

[1007] In another embodiment, the invention provides a method for operating a processor that includes, in response to executing a single arithmetic instruction, multiplying a first number by a second number, adding implicitly a partial result from a previously executed single arithmetic instruction, and adding a third number to generate a result that represents the first number multiplied by the second number

summed with the partial result and the third number. The method may further include storing a high order portion of the result as a next partial result into an extended carry register for use with execution of a subsequent single arithmetic instruction.

[1008] In still another embodiment, the invention provides a processor that includes an arithmetic circuit, the processor responsive to execution of a single arithmetic instruction to cause the arithmetic circuit to multiply a first and second number and add implicitly a high order portion of a partial result from a previously executed single arithmetic instruction, thereby generating a result that represents the first number multiplied by the second number summed with the high order portion of the partial result.

[1009] In still another embodiment, the invention provides a processor that includes an arithmetic circuit, the processor responsive to a single arithmetic instruction that upon execution thereof causes the arithmetic circuit to multiply a first number and a second number and add a third number and implicitly add a high order portion of a previous result from a previously executed single arithmetic instruction thereby generating a result that represents the first number multiplied with the second number, summed with the high order portion of the previous result and with the third number.

[1010] In still another embodiment, the invention provides a computer program product encoded on computer readable media. The computer program product includes a single arithmetic instruction causing a processor executing the single arithmetic instruction to multiply a first number by a second number and implicitly add a high order portion of a previously executed single arithmetic instruction to generate a result that represents the first number multiplied with the second number and summed with the a high order portion of a previously executed single arithmetic instruction. The single arithmetic instruction further causes the processor executing the instruction to keep a high order portion of the result for use with execution of a subsequent single arithmetic instruction.

[1011] In still another embodiment, the invention provides a computer program product encoded on computer readable media. The computer program product includes a single arithmetic instruction that causes a processor executing the single arithmetic instruction to multiply a first number by a second number, add implicitly a partial multiplication result from a previously executed single arithmetic instruction and a third number to generate a result that represents the first number multiplied by the second number summed with the partial multiplication result and summed with the third number. The single arithmetic instruction further causes the processor to store a high order portion of the result for use with execution of a subsequent single arithmetic instruction.

BRIEF DESCRIPTION OF THE DRAWINGS

[1012] The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings in which the use of the same reference symbols in different drawings indicates similar or identical items.

[1013] Fig. 1 illustrates an overview of a 1024 bit Montgomery modular multiplication.

[1014] Fig. 2 illustrates the add-chaining operation used for adding two multi-word integer values with automatic carry propagation.

[1015] Fig. 3 illustrates a multi-word multiplication.

[1016] Fig. 4A illustrates operation of the *umulxc* instruction.

[1017] Fig. 4B illustrates operation of the *umulxck* instruction.

[1018] Fig. 4C illustrates another embodiment of operation of the *umulxck* instruction.

[1019] Fig. 5 illustrates the calculation of a 64x1024 bit partial product and the accumulation with a previous partial product.

[1020] Fig. 6 illustrates an exemplary embodiment of the *umulxck* instruction that supports two threads.

[1021] Fig. 7 illustrates a multi-threaded data path with two functional units that in combination support the *umulxck* instruction for four threads.

[1022] Fig. 8 shows an embodiment in which the extended carry values can be exchanged between two functional units.

[1023] Fig. 9 shows an embodiment in which the extended carry register and the k register are readable and writeable.

[1024] Fig. 10 shows an example of a multiply-accumulate operation in an embodiment of the invention that utilizes an extended carry.

[1025] Fig. 11 shows an example of a multiply-accumulate operation in an embodiment of the invention that utilizes an extended carry in the redundant sum-and-carry representation.

[1026] Fig. 12 illustrates the inputs supplied to the carry look-ahead adder.

[1027] Fig. 12B illustrates an example using Booth encoding.

[1028] Fig. 13 shows a multiplier circuit structure that may be adapted for use in various embodiments of the invention.

[1029] Fig. 14 shows a multiplier circuit, that may be adapted for use in various embodiments of the invention, in which the adder circuit is implemented as two 64-bit adders.

[1030] Fig. 15 shows a multiply-and-accumulate circuit according to an embodiment of the invention.

[1031] Fig. 15A shows an exemplary implementation of an adder circuit.

[1032] Fig. 16 shows a multiply-and-accumulate circuit according to another embodiment of the invention.

[1033] Fig. 17 shows another embodiment of a multiply-and-accumulate circuit according to an embodiment of the invention, in which the extended carry is stored in a 64+64+2-bit representation.

[1034] Fig. 17A shows an exemplary implementation of an adder circuit.

[1035] Fig. 18 shows another embodiment of a multiply-and-accumulate circuit, in which the extended carry bits and carry registers are fed back and added in the Wallace tree.

[1036] Fig. 19 shows another embodiment of a multiply-and-accumulate circuit, in which the extended carry bits are fed back and added in the Wallace tree and the carry out bit from the adder circuit is fed back to the adder circuit.

[1037] Fig. 20 illustrates a multiply-and-accumulate circuit according to an embodiment of the invention implementing the *umulxck* and *bmulxck* instructions.

[1038] Fig. 21A shows an embodiment implementing the *umulxck* and *bmulxck* instructions in which the additions are performed in the Wallace tree for the feedback carry and sum bits and the extra term.

[1039] Fig. 21B illustrates a control circuit that allows various ones of the multiplier circuits described herein to be utilized for regular multiplications as well as multiplications involving the extended carry register. Further, it allows the circuit to be used when multiply-accumulate instructions are not necessarily on consecutive clocks.

[1040] Fig. 21C illustrates another embodiment implementing the *umulxck* and *bmulxck* instructions in which the additions are performed in the Wallace tree for the feedback carry and sum bits and the extra term.

[1041] Fig. 22A illustrates a full adder.

[1042] Fig. 22B illustrates a 4 to 2 compressor.

[1043] Fig. 22C illustrates a 5 to 3 compressor.

[1044] Fig. 23A shows one implementation of an XOR gate.

[1045] Fig. 23B shows an implementation of a two input multiplexer.

[1046] Fig. 23C shows a majority gate made out of NAND gates.

[1047] Fig. 23D shows a mul majority gate.

[1048] Fig. 24 shows an example of a Wallace tree column for a 64x64 multiplier with Booth encoding.

[1049] Fig. 25 shows another example of a Wallace tree column for a 64x64 multiplier with Booth encoding.

[1050] Fig. 26 shows another example of a Wallace tree column for a 64x64 multiplier with Booth encoding.

[1051] Fig. 27 illustrates an exemplary multiply-accumulate circuit according to an embodiment of the invention for implementing the *bmulxc* and *umulxc* instructions utilizing three feedback terms.

[1052] Fig. 28 illustrates an exemplary embodiment of a Wallace tree column used in a 64x64 multiplier with Booth encoding that supports the *umulxc* and *bmulxc* instructions for both integer multiply-accumulate and XOR multiply-accumulate.

[1053] Fig. 29 illustrates another exemplary embodiment of a Wallace tree column used in a 64x64 multiplier with Booth encoding that supports the *umulxc* and *bmulxc* instructions for both integer multiply-accumulate and XOR multiply-accumulate.

[1054] Fig. 30 illustrates an exemplary embodiment of a Wallace tree column used in a 64x64 multiplier with Booth encoding that supports the *umulxck* and *bmulxck* instructions for both integer multiply-accumulate and XOR multiply-accumulate.

[1055] Fig. 31 illustrates another embodiment of a Wallace tree column used in a 64x64 multiplier with Booth encoding that supports the *umulxck* and *bmulxck* instructions for both integer multiply-accumulate and XOR multiply-accumulate.

[1056] Fig. 32 illustrates another embodiment of a Wallace tree column used in a 64x64 multiplier with Booth encoding that supports the *umulxck* and *bmulxck* instructions for both integer multiply-accumulate and XOR multiply-accumulate.

DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

[1057] Multi-word multiplications and additions may be computed using many word-sized multiply, add, and shift operations. As shown in Fig. 1, a 1024-bit integer X, for example, can be represented with sixteen 64-bit words X=(x15, ..., x1, x0). That form is commonly referred to as multi-precision representation. The more efficiently a multi-word operation can be performed, the better the public-key performance will be. Adding capabilities to a general purpose processor to speed up multi-word operations is a key to accelerating public-key computations, and *add-chaining* and *multiply-chaining* are two of the capabilities needed.

[1058] A single architecture with a coherent instruction set for cryptographic support can reduce the software deployment cost and shorten the time-to-market. It thereby enables earlier adoption of the cryptographic capabilities found in the new systems. Implementing new instruction capability may be used to provide more efficient support for cryptographic computations. In one or more embodiments of the invention one or more instructions are provided that multiply two n bit numbers together and save the high order bits of the result in an extended carry register for use by the next multiply operation.

[1059] Prior to explaining that new instruction capability, additional details on Montgomery multiplication will be provided. As an example, 1024-bit Montgomery modular multiplication will be introduced to better understand the complexity involved in such a computation, which will facilitate an understanding of operation and advantages of the present invention. Montgomery modular multiplication is an operation commonly used to efficiently implement RSA.

[1060] Modular multiplication requires multiplication and reduction, the latter reducing the multiplication result back to the size of the input operands. A plain modular multiplication reduces the most significant bits whereas Montgomery modular multiplication reduces the least significant bits, which can be done much more efficiently. Fig. 1 illustrates the computation of a 1024-bit Montgomery modular multiplication split up into 32 rows representing multi-word multiplications and multi-word additions. The Montgomery method interleaves partial product generation and reduction. There are 16 rows y0*X....y15*X that show the multi-word multiplications resulting in the partial products. Interleaved with these rows are 16 additional rows of multi-word multiplications resulting in the reduction terms n0*P...n15*P. These terms are multiples of the modulus P that are chosen in a way such that the least significant word of the sum of the accumulated partial products at each intermediate stage yields zero.

[1061] The architecture of a particular processor can significantly affect the overall performance with respect to Montgomery modular multiplication. For example, it is advantageous if an architecture can support the new instructions proposed herein with a fully pipelined multiplier. For many processor architectures, the number of multiplications determines the upper performance bound for Montgomery exponentiation. A 1024-bit Montgomery modular multiplication requires $2*(16^2)+16=528$ 64-bit multiplications. Therefore, improving multiplication throughput generally improves the performance of Montgomery modular exponentiation. Multiplication latency can be hidden as long as sufficient registers are available to store intermediate results.

[1062] In addition to multiplications, a 1024-bit Montgomery modular multiplication requires $4*(16^2)+3*16-1=1071$ 64-bit additions (including carry propagation) to accumulate partial products. If additions take less time than multiplications and can be executed in parallel to multiplications, their cost can be hidden. However, low addition throughput and costly carry propagation can negatively contribute to the performance of Montgomery modular multiplication and can even determine its upper bound.

[1063] A 1024-bit Montgomery modular multiplication requires the generation and accumulation of 512 64-bit partial products yielding a 1024-bit end result . Besides 16 registers for accumulating the end result, a number of registers are needed for storing multiplier and multiplicand operands, constants, pointers and intermediate results. Although generation and accumulation of partial products can be pipelined, computational dependencies and instruction latencies can significantly increase register demand. Diminished performance can be exhibited if load and store operations are required to transfer intermediate results to and from cache memory.

[1064] Note that for enhanced performance of Montgomery computation, it is desirable to have registers for at least one 1024-bit operand and for accumulating an intermediate 1024+64-bit result. Smaller register files require frequent load and store operations to supply the operand to be multiplied in a 64x1024-bit multiplication and to spill some data back into the memory. If additional load and store operations are necessary to transfer intermediate results, the memory bandwidth between level-1 cache and register file can become a performance bottleneck. Efficient implementations of Montgomery modular multiplication preferably utilize the parallel execution of multiplications, additions and load/store operations. Pipeline dependencies that prohibit the parallel execution of these operations can significantly impact the performance.

[1065] Fig. 2 illustrates the add-chaining operation used for adding two multi-word integer values with automatic carry propagation. That is, the carry out of the previous addition operation is automatically propagated into the next addition operation. For example, assume there are three addition instructions *addcc, addxc*, and *adxccc*. Assume that *addcc* is an addition operation that produces a carry (reflected in location cc, which may be a condition code register or other location in the processor) but does not consume a carry, *addxc* is an addition operation that consumes a carry (the carry being based on the value in location cc), but does not produce one, and *adxccc* is an addition operation that both produces and consumes a carry. Adding two large (e.g., 1024 bit) integers X=(x15, ..., x1, x0) and Y=(y15, ...y1, y0) can be done very efficiently using 17 instructions whereby the carryout bit of one addition is automatically propagated into the next addition.

```
s0 = addcc x0, y0;
s1 = addxccc x1, y1;
          . . .
          . . .
          . . .
s15= addxccc x15, y15;
s16= addxc 0,0          // catch the last carryout bit.
```

[1066] The multiply-chaining operation is for computing multi-word multiplication with automatic carry propagation. Fig. 3 shows an example of a multi-word multiplication *y0*X* where *y0* is a 64-bit integer and X is a 1024-bit integer X=(x15, ..., x1, x0). The multiplication will be explained assuming the instructions *umulxhi and mulx* are available. The instruction *umulxhi (rs1, rs2, rd)* is an unsigned operation that multiplies two 64 bit numbers specified as the source operands *rs1* and *rs2* and places the high 64 bits of the 128 bit result in the destination register *rd*. The instruction *mulx (rs1, rs2, rd)* multiplies two 64 bit numbers specified in the source operands *rs1* and *rs2* and places the low 64 bits of the 128 bit result in the destination register *rd*. Assuming such instructions, the computation *y0*X*, illustrated in Fig. 3, can be carried out in the following instruction steps, where h0 represents the high 64 bit result of multiplying x0 and y0 and l0 represents the lower 64 bit result of multiplying x0 and y0:

```
h0  = umulxhi  x0, y0;        l0  = mulx  x0, y0;
h1  = umulxhi  x1, y0;        l1  = mulx  x1, y0;
                 . . .
                 . . .
                 . . .
h15 = umulxhi  x15, y0;       l15 = mulx  x15, y0;
r0 = l0;
r1 = addcc h0, l1;
r2 = addxccc h1, l2;  // catch the carryout bit.
                 . . .
                 . . .
                 . . .
r15= addxccc h14, l15;
r16= addxc h15,0;      //
```

[1067] Note that the upper 64-bits, for example, *h0*, of a 128-bit partial product *x0*y0* is manually propagated into the next partial product *x1*y0* using an *addcc* instruction. That process is typically slow because the output is delayed by the multiplier latency, which may be, e.g., an 8-cycle latency in the case of an exemplary processor. The present invention provides a more efficient technique for efficiently handling the propagation of the upper 64-bits of a 128-bit product into a next operation.

[1068] Referring again to Fig. 1, three modes of operation may be used to accelerate Montgomery modular multiplication - the modes are called "add-chaining", "multiply-chaining", and "multiply-accumulate-chaining".

[1069] In one embodiment of the invention an unsigned multiplication using an extended carry register (the instruction *umulxc*) performs a multiply-and-accumulate computation and returns the lower 64-bits of (rs1*rs2 + previous extended carry) and saves the upper 64 bits of the result in an extended carry register to be used by the next multiply operation. The lower 64 bits of the multiply-and-accumulate result are referred to herein as the product and the upper 64 bits are referred to herein as the extended carry. While traditionally an add carryout is only 1 bit and is contained in location cc, the instruction *umulxc* defines a 64-bit extended carry register (*exc*) that contains the extended carry bits. The extended carry register enables the automatic propagation of the carryout bits in a multiply-chaining operation such that a multi-word multiplication can be executed in consecutive instructions.

[1070] The *umulxc* instruction is illustrated in Fig. 4A and summarized in Table 1 below. The result register rd receives the lower n bits [n-1:0] 401 of (rs1*rs2 + previous extended carry saved in the extended carry register (*exc*) 403) and saves the upper n bits [2n-1:n] 405 of (rs1*rs2 + previous *exc*) in the extended carry register (*exc*) 403 for use in subsequent computations. The exc value, although saved from the most significant n bits of the result of one operation, is added into the least significant n bits of the next operation. Note that in the implementation illustrated in Fig. 4A the *exc* register is a register that is logically local to the multiplier, and may be implemented as a special register so that, even though not a general purpose register such as those specified by rs1, rs2 and rd, the *exc* register can be accessed in association with, e.g., saving and restoring the *exc* register in association with context switches. The *exc* register is used to propagate an n bit extended carry per multiplication. The source operands rs1, rs2, the destination register rd, and the extended carry register are assumed to be n bits. In an exemplary embodiment, such as the embodiment described in Table 1, n=64.

Table 1

Instruction	Description
umulxc rs1, rs2, rd	Computes rd = lower 64 bits of (rs1*rs2 + previous carry saved in *exc*) and the upper 64 bits of (rs1*rs2 + previous *exc*) are saved in *exc* for use in subsequent computations.

[1071] Referring again to Fig. 3, the multiplication of a 64-bit integer, y0, and a 1024-bit large integer X= (x15, ...,x1, x0) can be done using the *umulxc* instruction in the following 18 steps:

```
umulxc 0,0; // clear extended-carry register exc first
r0 = umulxc  y0, x0;
r1 = umulxc y0, x1;

. . .

. . .

. . .
r15 = umulxc y0, x15;
r16 = umulxc 0, 0;    // save the last carryout bits.
```

[1072] Referring to Fig. 5, in one implementation, a multiplication algorithm may use a sequence of *umulxc* instructions to compute a row (e.g. *y0*X*) and a sequence of add instructions, e.g., *addcc*, *adxccc*, to accumulate two rows. Note that the first instruction umulxc 0,0; clears the extended-carry register. Alternatively, an instruction can be defined that produces, but does not consume an extended carry, and can be utilized to compute r0, to eliminate the need for an explicit instruction clearing the extended carry register.

[1073] According to another embodiment of the invention an instruction, *umulxck,* effectively combines both multiply and accumulate operations. In addition to computing a row *y0*X,* the *umulxck* instruction also allows for accumulating an additional row *S=(s15, ..., s0)* implicitly without requiring additional add (e.g., *adxccc)* operations. The *umulxck* instruction is illustrated in Fig. 4B and summarized in Table 2 below. The result register rd receives the lower n bits 407 of (rs1*k + previous extended carry saved in the extended carry register (*exc*) 403 + rs2). The extended carry register 403 receives the upper n bits 405 for use in subsequent computations. As with the *umulxc* instruction, the exc value, although saved from the most significant n bits of the result of one operation, is added into the least significant n bits of the next operation. The register rs2 is used to provide the words of the accumulated partial products. Note that in the implementation illustrated in Fig. 4B the extended carry register is logically local to the multiplier and is used to propagate an n-bit extended carry per multiplication. The *exc* register illustrated in Fig. 4B may be implemented as a special register so that, even though not a general purpose register such as those specified by rs1, rs2 and rd, the *exc* register can be accessed in association with, e.g., saving and restoring the *exc* register in association with context switches. The source operands rs1, rs2, the destination register rd, the extended carry register and the k register are assumed to be n bits. In the embodiment described in Table 2, n=64.

Table 2

Instruction	Description
umulxck rs1, rs2, rd	Computes rd = lower 64 bits of (rs1*k + rs2 + previous carry saved in *exc*) and saves the upper 64 bits of (rs1*k + rs2 + previous *exc*) in *exc* for use in subsequent computations.

[1074] In the embodiment illustrated in Fig. 4B, the *umulxck* instruction uses a logically local register k rather than a general-purpose register for two reasons. First, some instruction formats, e.g. the SPARC™ instruction format allows for specifying only two source operands. Secondly, referring again to the multiply-chaining shown in Fig. 1, one operand (y0) remains constant throughout the computation of an entire partial product and, therefore, can be kept in a local register that is initialized only once for every partial product. The *k* register illustrated in Fig. 4B may be implemented as a special register so that, even though not a general purpose register such as those specified by rs1, rs2 and rd, the *k* register can be accessed in association with, e.g., saving and restoring the register in association with context switches.

[1075] Fig. 4C shows an alternative embodiment of an implementation of the *umulxck* instruction having a single summing node 410.

[1076] In other embodiments, the register k may be explicitly specified as one of three source operands *umulxck (rs1, rs2, rs3, rd)* to perform rs1*rs2 + rs3 + *exc* and store the low order portion of the result in the result register rd. In still other embodiments, one of the source registers may be identical with the destination register. For example, the instruction *umulxck rs1, rs2, rs3* executes rs1*rs2 + rs3 + *exc* and stores the result in rs3. In still another embodiment, the register k may implicitly be a specific general purpose register, e.g. the register r0. For example, the instruction *umulxck rs1, rs2, rd* performs rs1*r0 + rs2 + *exc* and stores the result in the result register rd. If rs1 is specified to be the register r0, a square operation will be performed.

[1077] The *umulxck* instruction is an efficient way to support public-key computations. Back-to-back scheduling of multi-word multiplications and accumulations is often difficult using today's instruction sets due to the multiplier latency. The *umulxck* instruction performs the *multiply-accumulate-chaining* operation which combines add-chaining and multiply-chaining in one operation and avoids the multiplier latency. Using the *umulxck* instruction, and referring again to Fig. 5, the calculation of a 64x1024 bit partial product (y0 * X) and the accumulation with a previous partial product (s16:s0), can be accomplished in the following 20 instructions:

```
set register k=y0;
umulxck 0,0;              // clear extended-carry register exc first
r0 = umulxck x0, s0;

. . .

. . .

. . .
r15 = umulxck x15, s15;
r16 = umulxck 0, s16;     // catch 64 carryout bits
r17 = umulxck 0, 0;       // catch last carryout bit
```

[1078] Note that the final *umulxck* instruction illustrated above clears the extended carry register so that in a chained partial product calculation, the first *umulxc* instruction illustrated above is unnecessary after the first partial product calculation.

[1079] After the local k register is set to y0, the extended carry register is explicitly cleared (*umulxck* 0,0). Alternatively, the second *umulxck* instruction (*umulsck* x0, s0) could be replaced by a multiply instruction that produces but does not consume an extended carry.

[1080] Thus, several embodiments of instructions and implementations have been described that accelerate multiply-chaining. An architecture only needs to support one or the other instruction. In addition to increased performance, multiply-accumulate-chaining has the advantage that it only keeps the multiplier busy whereas other functional units of a processor are unused. That is, in a multi-threaded implementation the latter units could be used by other threads.

[1081] It is worth noting that neither the *umulxck* nor the *umulxc* instruction produce an overflow. This is because a 64x64 product is not greater than $(2^{64}-1)(2^{64}-1)$ in magnitude and adding two 64-bit integers to the 128-bit product will not cause an overflow.

$$(2^{64}-1)(2^{64}-1) + (2^{64}-1) + (2^{64}-1) < 2^{128}$$

[1082] Elliptic curve public-key cryptographic systems are defined over two types of arithmetic fields, integer fields and binary polynomial fields. The arithmetic operations in both types of fields are similar. An addition in a binary polynomial field is a bit-wise exclusive-or operation. A multiplication, referred to herein as "XOR multiply", is similar to an integer multiply except that partial products are summed with bit-wise exclusive-or operations. An execution unit that supports both integer and XOR multiplications is described in Application No. 10/354,354, filed January 30, 2003, entitled MULTIPLY EXECUTION UNIT FOR PERFORMING INTEGER AND XOR MULTIPLICATION, naming Rarick et al. as inventors, which application is incorporated herein by reference in its entirety.

[1083] Providing instruction set and hardware support today for XOR multiply operations further enhances performance for elliptic curve cryptographic applications. The integer multiplier unit can be readily modified to perform an XOR multiply. That function can be implemented in the block diagrams illustrated in Figs. 4A, 4B and 4C where the arithmetic operations are XOR multiplies and XOR additions.

[1084] Tables 3 and 4 summarize two instructions *bmulxc* and *bmulxck* that are utilized to perform XOR multiplications for use in binary polynomial field operations, and correspond to *umulxc* and *umulxck*, respectively. The instruction *bmulxc (rs1, rs2, rd)* stores in destination register rd, the lower n bits of (rs1^*rs2 ^ previous *exc*), where rs1^*rs2 refers to the XOR multiply and the symbol ^ refers to addition in a binary polynomial field (a bit-wise XOR operation). The instruction saves the upper n-1 bits of (rs1^*rs2 ^ previous *exc*) in the extended carry register *exc*. The source operands rs1, rs2, the destination register rd, and the extended carry register are assumed to be n bits. In the embodiment illustrated in Table 3, n=64 bits.

Table 3

Instruction	Description
bmulxc rs1, rs2, rd	computes rd = lower 64 bits of (rs1^*rs2 ^ previous *exc*) and saves the upper 63 bits of (rs1^*rs2 ^ previous *exc*) in register *exc*. Here the multiply, rs1^*rs2, refers to the XOR multiply.

[1085] The instruction *bmulxck* stores in destination register rd, the lower n bits of (rs1^*k ^ rs2 ^ previous *exc*), where rs1^*k refers to the XOR multiply and the symbol ^ refers to addition in a binary polynomial field (a bit-wise XOR operation). The instruction saves the upper n-1 bits of (rs1^*k ^ rs2 ^ previous *exc*) in the extended carry register *exc*. The source operands rs1, rs2, the destination register rd, the extended carry register, and the k register are assumed to be n bits. In the embodiments illustrated in Tables 4, n = 64.

Table 4

Instruction	Description
bmulxck rs1, rs2, rd	computes rd = lower 64 bits of (rs1^*k ^ rs2 ^ previous exc) and saves the upper 63 bits of the result in *exc* for use in subsequent computations. Here the multiply, rs1^*k, refers to the XOR multiply.

[1086] In current processor implementations, it is common for multiple threads to be running concurrently on a single processor. That leads to the possibility that multiple threads can use the multipliers described in Figs. 4A, 4B, and 4C. However, because in an embodiment local multiplier registers, rather than general purpose registers, are used for the extended carry register and register k (see Fig. 4B and 4C), there should be a way to ensure that when switching between threads, the information in the local multiplier registers is not lost. Referring to Fig. 6, an exemplary embodiment shows a data path that supports two threads. Local registers are provided for each thread in functional unit 600. Thus, an extended carry register 601 is provided for thread 0 and an extended carry register 603 is provided for thread 1. Similarly, a local k register 605 is provided for thread 0 and a local k register 607 is provided for thread 1.

[1087] Fig. 7 shows a multi-threaded data path with two functional units 701 and 703 that in combination support four threads. In the embodiment illustrated in Fig. 7, threads are preassigned to functional units so that the extended carry value is always locally available. Threads 0 and 1 are assigned to functional unit 701. Thus, functional unit 701 includes extended carry registers 705 and 707 and k registers 709 and 711. Functional unit 703 includes extended carry registers 715 and 717 and k registers 719 and 721. The particular extended carry register and the k register utilized are selected according to the thread being processed by the functional unit. The registers 723, 725, 727, and 729 are not logically local registers in that upon execution they are loaded with the contents of the general purpose registers specified as source operands rs1 and rs2.

[1088] Fig. 8 shows an embodiment in which the extended carry values can be exchanged between two functional units 801 (multiplier 0) and 803 (multiplier 1). With the embodiment illustrated in Fig. 8, threads are not preassigned to functional units and can be executed on either functional unit. For example, if a thread is reassigned from one to the other functional unit, multiplixer logic 805 and 807 can select the extended carry value of the previous multiplication available for accumulation with the newly computed product from the extended carry registers in either multiplier 0 or multiplier 1. Thus, e.g., if a thread 0 had been executing on functional unit 801 (multiplier 0), and is reassigned to functional unit 803 (multiplier 1) for the subsequent multiplication, the multiplexer logic 809 on functional unit 801 selects the extended carry register 811 associated with thread 0 and multiplexer logic 807 in functional unit 803 is controlled to select the other multiplier extended carry value supplied on node 812. Multiplexer 808 selects the extended carry register supplied to multiplexer 805. In functional unit 803, multiplexer 814 selects the extended carry register to be supplied to functional unit 801 and multiplexer 816 selects the extended carry register to be supplied to multiplexer 807. Note that copies of the k registers for each of the four threads are provided in each functional unit in the illustrated embodiment.

[1089] In another embodiment (not shown in Fig. 8) the functional units can selectively store the high order bits of the addition from either functional unit into a selected one of the extended carry register(s) by using multiplexer logic on the input side of the extended carry register(s) to select which functional unit should be the source of the extended carry register.

[1090] Fig. 9 shows an embodiment in which the extended carry register 901 and the k register 903 are readable and writeable. That capability is needed, in particular, for implementing context switches. To reduce overhead for context switches in the illustrated embodiment, dirty bits 905 and 907 are associated with these registers. A dirty bit is set the first time the corresponding register is written after a context switch. The dirty bit is reset at context switch. A register is saved on context switch only if the dirty bit is set. In another embodiment, the extended carry register 901 and the k register 903 share a dirty bit since a write to k is likely to be followed by a write to the extended carry register.

[1091] The instructions capability described herein is intended to accelerate multi-word multiplications through multiply-chaining. Referring to Fig. 1, in an embodiment of the invention using *umulxck*, it takes a total of 35 instructions to compute and accumulate two rows, i.e. a partial product and a reduction term, of a Montgomery modular multiplication:

```
Multi-word multiplication and accumulation  S = S + xi * Y  - 17 instructions.
Multi-word multiplication and accumulation  S = S + ni * P  - 18 instructions.
                                                            --------------------
                                                            35 umulxck instructions
```

[1092] Appendix A illustrates sample pseudo code for a 256x256 multiplication using the *umulxck* instruction. Appendix A helps illustrate the simplicity offered by use of this new instruction capability. The pseudo code shows a 256x256-bit multiplication of two arrays A=(a3, ..,a0) and B=(b3, ...,b0). Note that the cycle count depends on the instruction latencies of the actual implementation. The value for k can be first loaded into a general-purpose register and then be moved into the k register (see Appendix A).

[1093] While Figs. 4A, 4B, 4C, 6, 7, and 8 provide various embodiments of the multiplier and addition logic utilizing the *exc* register and *k* register (where applicable), other embodiments to efficiently implement the multiply and add operations may incorporate the additions required to add the high order bits of the result from the previous *umulxc* (or *umulxck*) instruction into the addition operations performed in association with summing of the partial products that are part of multiplication of the two operands in the current *umulxc* (or *umulxck*) instruction. In fact, the *umulxc* and *umulxck* instructions can usually be implemented so that they are approximately as fast as an ordinary multiply instruction. This is because an ordinary multiply instruction needs to add many partial products, which is commonly implemented with a tree of carry save adders. Including the extended carry register for *umulxc*, or both the extended carry register and rs2 for *umulxck*, as additional terms in the tree does not significantly affect the time to generate the product. Note that in such an implementation the addition(s) specified in Figs. 4A, 4B, 4C (as well as Figures 6, 7, 8, and 9) are not performed after the multiplication but are an integral part of the multiplication.

[1094] There are many techniques used to efficiently and rapidly perform multiplication operations. Fast integer multipliers may be constructed that utilize carry save adders, full adders, 4 to 2 compressors, and 5 to 3 compressors. In one embodiment of the invention, a multiply execution unit performs both integer and XOR multiplication so that arithmetic operations for binary polynomial fields can be supported as well as integer arithmetic. An execution unit that supports both integer and XOR multiplications is described in Application No. 10/354,354. filed January 30, 2003, entitled MULTIPLY EXECUTION UNIT FOR PERFORMING INTEGER AND XOR MULTIPLICATION, naming Rarick et al. as inventors, which application was previously incorporated herein by reference in its entirety.

[1095] Referring back to Fig. 4A, illustrated is the computation of the result (exc, rd) = rs1 * rs2 +exc, where the upper n bits of the multiply-accumulate operation are stored in the extended carry register exc and the lower n bits are stored in the result register rd . That operation suggests that all partial products of the multiplication and the extended carry register are added before they are stored in the extended carry register exc and the result register rd. However, hardware implementations of multipliers often choose to generate an intermediate result in a redundant number representation that requires one more addition to generate the final result. Fig. 10 illustrates that operation giving an example of a 4x4 bit multiply-accumulate operation, where the multiplication result (exc, rd) = (X3, X2, X1, X0) * (Y3, Y2, Y1, Y0) + (EX3, EX2, EX1, EX0), where (EX3, EX2, EX1, EX0) are the high order results of a previous multiply-accumulate operation. As can be seen in Fig. 10, the multiplication result (S6, S5, S4, S3, S2, S1, S0, C7, C6, C5, C4, C3, C2, C1) is provided in a form that requires an addition to obtain the final result. Note that the high order bits of the result (EX3, EX2, EX1, EX0) are those bits that are utilized in a subsequent multiply-accumulate operation.

[1096] In one embodiment, the partial products P00 . . . P33 and the extended carry bits (EX3, EX2, EX1, EX0) are added using a Wallace tree. A Wallace tree is a structure of full adders that generates a result having two sets of numbers, designated in Fig. 10 as sum outputs S6 to S0 and carry outputs C7 to C1. The final result shown in Fig. 10 (EX3, EX2, EX1, EX0, RD3, RD2, RD1, RD0) can be computed from the sum and carry outputs of the Wallace tree in an adder circuit. A common implementation of that adder circuit utilizes a carry look-ahead adder in order to propagate the carries involved in the addition efficiently.

[1097] Instead of storing the final high order bits of the result (EX3, EX2, EX1, EX0) which incurs the delay associated with the addition, only the low order bits required for the destination register (RD3, RD2, RD1, RD0) in the example shown in Fig. 10, need to be sent to the carry look-ahead adder, while the high order bits of the result can be kept in the sum-and-carry representation. By keeping the high order portion in the redundant representation, the appropriate sum and carry bits representing the high order portion of the result can be fed back more quickly into the Wallace tree.

[1098] Fig. 11 shows an example of a multiply-accumulate operation in an embodiment of the invention that utilizes an extended carry in the redundant sum-and-carry representation. The partial products P00 to P33 are inputs at the start of the Wallace tree. The previous high order sum output bits of the Wallace tree (S6, S5, S4) and the previous carry output bits (C7, C6, C5, C4) are input back into the Wallace tree. The new sum outputs of the Wallace tree (S6, S5, S4, S3, S2, S1, S0) and the new carry outputs (C7, C6, C5, C4, C3, C2, C1) are generated using the previous carry and sum output bits, thus efficiently feeding back the extended carry bits into the Wallace tree. Thus, the extended carry bits are summed with the partial products of the multiplication to achieve much greater efficiency than if done in a separate addition operation after the multiplication result is calculated in the carry look-ahead adder (CLA). Note that a carry bit CC4 that is necessary to add into the high order carry and sum outputs to fully represent the high order result of the multiplication is not fed back into the Wallace tree but is fed back into the carry look-ahead adder instead.

[1099] Referring to Fig. 12 the inputs to the carry look-ahead adder are shown. The carry look-ahead adder changes the redundant representation into the final result. The inputs to the carry look-ahead adder are shown to be the low order sum bits from the Wallace tree (S3, S2, S1, S0) and the low order carry bits from the Wallace tree (C3, C2, C1). Note that a carry out from the previous carry-look ahead addition would normally be propagated by adding the carry out with the lowest order bits S4 and C4 Wallace tree outputs. However, rather than feeding that carry out into the Wallace tree, which would require waiting for completion of the carry look-ahead addition operation, that carry out bit (CC4 in Fig. 12) is simply added into the carry look-ahead adder along with the outputs of the Wallace tree.

[1100] Note that the number and position of the sum bits and the carry bits can vary widely. That is so, in part, because a Wallace tree can take many different forms. Also, whether or not Booth encoding is used can also have an effect. For a k by k multiplication, the maximum bit configuration as output form the carry save adder is:

sum output:	S[2k-1], S[2k-2], ..., S[3], S[2], S[1], S[0]
carry output:	C[2k], C[2k-1], C[2k-2], ..., C[3], C[2], C[1], C[0]

[1101] Note that the C[2k] bit could be called S[2k] instead. One or more of the carry bits and one or more of the sum bits could be known to always be zero, any such bits being determined by such factors as the details of the Wallace tree and whether Booth encoding is used. For example, if Booth encoding is not used, C[2k] is known to be always zero. If Booth encoding is not used and no half adders are used, then one can also have S[2k-1] and C[2k-1] be zero. If Booth encoding is not used, then one can have one of S[2k-1] and C[2k-1] be known zero. The bit that is known zero is a matter of naming convention. Often, one can have several, e.g., up to half a dozen or so, of the least significant bits (e.g., C[0], C[1], C[2], ...) be known to be zero. These could be either sum or carry bits (or a mixture), depending on the naming convention. There can be special conditions where about a quarter of the above bits could be known to be zero. Thus, many conditions exist that effect the specific number of bits and which bits are known zeros.

[1102] Referring to Fig. 12B, the example shown uses Booth encoding and multiplies X = [X7, X6, X5, X4, X3, X2, X1, X0] by Y = [Y7, Y6, Y5, Y4, Y3, Y2, Y1, Y0] and adds in term Z = [Z7, Z6, Z5, Z4, Z3, Z2, Z1, Z0], using the *umulxck* instruction. The "partial product" terms produced by the Booth encoding muxes are designated by D, E, F, G, and H in Fig. 12B. The sign of each of these terms is SD, SE, SF, and SG, with H known to be non-negative. The first five lines shown in Fig. 12B are from the Booth encoding. The next two lines are the feedback of the high order sum and carry bits from the previous execution of the *umulxck* instruction, and the last line is the Z term to be added in. [S7:S0] and [C7:C0] are provided to the carry look-ahead adder. C16 is the sign bit, as one of the two terms (S and C) that are fed back may be negative when Booth encoding is used. The multiple occurrences of C16 when fed back are due to sign extension.

[1103] Referring again to Fig. 9, the implementation shown utilizes added wires and multiplexer circuitry to allow access for loading and storing the exc register, e.g., when a context switch is needed. However, the additional hardware requirements to achieve that access may be undesirable in particular embodiments. In another embodiment, the extended carry register exc can be loaded and stored using *umulxc/umulxck* instructions as explained further below. When a context switch is needed, it is desirable to save the non-redundant extended carry representation (e.g., ex3, ex2, ex1, ex0) instead of the larger redundant extended carry representation (e.g., S6, S5, S4, C7, C6, C5, C4, CC4), shown, e.g., in Fig. 11. The code that performs the context switch, after it has saved the general purpose registers, can perform an *umulxc* or *umulxck* instruction to load and/or store the extended carry register as part of that context switch. In one embodiment, the processor executes a *umulxc* or *umulxck* instruction that multiplies zero by zero (and adds zero for *umulxck*). The output result (rd) will be the non-redundant (ex3, ex2, ex1, ex0) value to be saved for the context switch. This also sets the exc register (i.e., the sum, carry, and CC4) to zero, which is needed before restoring the new context. In order to restore the exc value for the new context, two cases need to be considered.

[1104] If the *umulxck* instruction is implemented, then the current exc value can be obtained and the new exc context value restored at the same time. Let v be the new exc value. Assume the *umulxck* instruction is executed with rs1=v, rs2=v and k=$(2^n)-1$ (all bits on). This computes $v^* (2^n-1) + v +$ current exc $= v^* 2^n +$ current exc. Thus, the current exc is output to register rd and v becomes the new exc value.

[1105] If the *umulxck* instruction is not implemented, the *umulxc* instruction can be used to implement load and/or store operations of the extended carry register as part of the context switch. In this case note that the value of v can never exceed 2^n-2. That is so because the maximum value computable is all bits on times all bits on, plus the previous exc value. If the previous exc value is assumed to be $(2^n)-2$, then

$$\begin{aligned}(2^n-1) * (2^n-1) + (2^n-2) &= 2^{2n} - 2^n - 2^n + 1 + 2^n - 2 \\ &= 2^{2n} - 2*(2^n) \quad + 2^n - 1 \\ &= 2^n * (2^n - 2) \quad + 2^n - 1 \\ &= 2^n * (\text{new exc value}) + 2^n - 1\end{aligned}$$

and so the new exc value can never exceed $2^n - 2$ if the old value doesn't exceed that value. Hence, a larger value can not be achieved.

[1106] As stated above, the current extended carry value (exc) may be obtained by execution of an *umulxc* instruction that multiplies zero by zero. The output result (rd) will be the non-redundant extended carry value (ex3, ex2, ex1, ex0) to be saved for the context switch. That also sets the exc register (i.e. the sum, carry and CC4) to zero, which is needed for restoring the new context. The new context v may be restored by executing an *umulxc* instruction with rs1=v+1 and rs2=$(2^n)-1$ (all bits on). This computes

$$\begin{aligned}(v+1) * (2^n-1) &= v^* (2^n) + 2^n - v - 1 \\ &= (2^n) *v + (2^n - (v+1)) \\ &= (2^n) * (\text{restoring exc value}) + (2^n - (v+1)).\end{aligned}$$

Since $2^n - (v+1)$ is between 0 and $2^n - 1$, the value of v has been restored. Note that by saving and restoring the exc value in this manner, no extra instructions are needed and no extra data paths to or from the exc value need be provided, saving hardware resources.

[1107] The following pseudo code fragments illustrate how the extended carry can be stored and loaded on a 64-bit processor, i.e., n=64. The following pseudo code illustrates how the extended carry register can be stored (the value of the extended carry register retrieved) utilizing the *umulxc* instruction.

```
ldx 0, r1               // r1 = 0
umulxc r1, r1, r0       // (exc, r0) = 0 * 0 + exc, i.e.
                        // r0=old_exc, exc = 0
stx r0, [exc]
```

[1108] One embodiment of restoring exc values computes $(2^{64} - 1) * (\text{exc\_value} + 1) = (\text{exc\_value} * 2^{64}) + (2^{64} - (\text{exc\_value} + 1))$. The exc gets exc_value and r2 gets $2^{64} - (\text{exc\_value} + 1)$, which is to be ignored. Assume that the current value of the extended carry register exc is zero:

```
ldx [exc], r0                // r0 = exc_value with 0 <=
                             // exc_value <= 2^64 - 2 < 2^64 - 1
ldx 0xFFFFFFFFFFFFFFFF, r1   // r1 = 0xFFFFFFFFFFFFFFFF = 2^64 - 1
add r0, 1, r0                // r0 = exc_value + 1 with 0 <
                             // exc_value + 1 < 2^64
umulxc r0, r1, r2            // set exc to exc_value, ignore r2
```

[1109] Another embodiment of restoring exc_values computes $(2^{64} - 1) * \text{exc\_value} + 2^{64} - 2 = (\text{exc\_value} * 2^{64}) + (2^{64} - \text{exc\_value} - 2)$. Thus, the extended carry register exc can be loaded with the following pseudo code:

```
ldx [exc], r0                // r0 = exc_value
ldx 0xFFFFFFFFFFFFFFFF, r1   // r1 = 0xFFFFFFFFFFFFFFFF = 2^64 - 1
umulxc r1, r1, r2            // set exc = 0xFFFFFFFFFFFFFFFE =2^64
                             // - 2, ignore r2
umulxc r0, r1, r2            // set exc = r0, ignore r2
```

This scheme works for an arbitrary exc_value with $0 <= \text{exc\_value} <= 2^{64} - 2$ since $(\text{exc}, r2) = (2^{64} - 1) * \text{exc\_value} + 2^{64} - 2 = \text{exc\_value} * 2^{64} + (2^{64} - 2 - \text{exc\_value})$. The term $(2^{64} - 2 - \text{exc\_value})$ will never create a carry overflow into the upper 64 bits as long as $0 <= \text{exc\_value} <= 2^{64} - 2$. It can be mathematically shown that the definition of *umulxc* guarantees the extended carry register to always be in the range $0 <= \text{exc} <= 2^{64} - 2$ when exc is initially 0.

[1110] Processors that implement both integer and XOR multiply-accumulate instructions with a shared extended carry register can use *umulxc* or *umulxck* to store and restore the extended carry register. On processors that have separate extended carry registers for integer and XOR multiply-accumulate operations or that do not implement instructions *umulxc* and *umulxck*, the extended carry for XOR multiply-accumulate operations can be stored and restored with *bmulxc/bmulxck* instructions as described below. The value of the extended carry register for XOR multiply-accumulate can be obtained by executing a multiplication by 0. Using *bmulxc*, one can store the extended carry with the following pseudo code:

```
ldx   0, r1                         // r1 = 0
bmulxc r1, r1, r0                   // (exc, r0) = 0 *^ 0 ^ exc, i.e.
                                    // r0=old_exc, exc = 0
stx r0, [exc]                       // store r0
```

[1111] An alternative way to store the extended carry register exc with *umulxc* instructions adds 1 when storing it:

```
ldx 1, r1                  // r1 = 1
umulxc r, r1, r0           // (exc, r0) = 1 * 1 + exc,
                           // i.e., r0=old_exc+1, exc = 0
stx r0, [exc]
```

[1112] An alternative way to load exc with *umulxc* instructions subtracts 1 when loading it. Again, this approach works for 0 <= exc_value <= 2^{64} - 2.

```
ldx [exc], r0                      // r0 = exc_value + 1, assume exc = 0
ldx 0xFFFFFFFFFFFFFFFF, r1         // r1 = 0xFFFFFFFFFFFFFFFF = 2^64 - 1
umulxc r0, r1, r2                  // set exc = r0 - 1, ignore r2
```

[1113] When using the *umulxck* instruction, the following pseudo code can be used for storing exc:

```
ldx 0, r1                          // r1 = 0
umulxck r1, r1, r0                 // (exc, r0) = 0 * k + 0 + exc,
                                   // i.e., r0=old_exc, exc = 0
stx r0, [exc]
```

[1114] Likewise, the following pseudo code can be used for loading the extended carry register exc using the *umulxck* instruction:

```
ldx [exc], r0                      // r0 = exc_value, assume exc = 0
ldx 0xFFFFFFFFFFFFFFFF, r1         // r1 = 0xFFFFFFFFFFFFFFFF =
                                   // 2^64 - 1
mov r1, k                          // k = r1 = 2^64 -1
umulxck  r0, r1, r2                // set exc = r0, ignore r2
```

[1115] This scheme works for an arbitrary exc_value with $0 <= \text{exc\_value} <= 2^{64} - 1$ since $(\text{exc}, r2) = (2^{64} - 1) * \text{exc\_value} + 2^{64} - 1 + 0 = \text{exc\_value} * 2^{64} + (2^{64} - 1 - \text{exc\_value})$. That is, this scheme works for loading and storing arbitrary exc in the range $0 <= \text{exc} <= 2^{64} - 1$.

[1116] Similarly, the extended carry can be stored using the instruction *bmulxck*:

```
ldx 0, r1                      // r1 = 0
bmulxck r1, r1, r0             // (exc, r0) = 0 *^ k ^ 0 ^ exc,
                               // i.e., r0=old_exc, exc = 0
stx r0, [exc]                  // store r0
```

[1117] Note that the most significant bit in an n-bit extended carry register is always zero if XOR multiply-accumulate is used. That is due to the fact that the result of multiplying two n-bit binary polynomials can never be greater than 2n-1 bits, which can be split up into an n-bit result and an n-1-bit extended carry. Adding in one or more n-bit polynomials as in a *bmulxc/bmulxck* operation does not affect the size or value of the extended carry. For n=64, the extended carry register can be restored with an n-1-bit binary polynomial $\text{rest\_exc} = \text{rex\_62} * t^{62} + \text{rex\_61} * t^{61} + ... + \text{rex\_1} * t + \text{rex\_0}$ using *bmulxc* with the following pseudo code:

```
ldx [exc], r0             // r0 = rest_exc = rex_62 * t^62 + rex_61
                          // * t^61 + ... + rex_1 * t + rex_0
sll r0, 1, r0             // r0 = rex_62 * t^63 + rex_61 * t^62 +
                          // + rex_1 * t^2 + rex_0 * t
ldx 1, r1                 // r1 = 1
sll r1, 63, r1            // r1 = t^63
bmulxc r0, r1, r2         // set exc = rex_62 * t^62 + rex_61 *
                          // t^61 + ... + rex_1 * t + rex_0, r2 =
                          // old_exc
```

[1118] The extended carry value rest_exc being restored is first multiplied by t through a logical shift-left instruction (sll) and subsequently multiplied by t^{63} resulting in $(\text{exc}, r2) = \text{rest\_exc} *\wedge t^{64} \wedge \text{old\_exc}$. Note that restoring exc using *bmulxc* also reads out the previous value of the extended carry register exc. Thus, both can be done at this same time. That is, the operation performed is $(t*(\text{rest\_exc}) * t^{n-1} \wedge (\text{old\_exc})$.

[1119] The operation $(t*(\text{rest\_exc}) * t^{n-1}) = \text{rest\_exc} * t^{n}$ restores the rest_exc value without affecting the least significant n bits, and so the previous exc value is correctly output. The instruction *bmulxck* can be used by having the term being added in be zero.

[1120] Similarly, *bmulxck* can be used for restoring the extended carry:

```
ldx [exc], r0               // r0 = rest_exc = rex_62 * $t^{62}$ + rex_61 *
                            // $t^{61}$ + ... + rex_1 * t + rex_0
sll r0, 1, r0               // r0 = rex_62 * $t^{63}$ + rex_61 * $t^{62}$ + ... +
                            // rex_1 * $t^2$ + rex_0 * t
ldx 1, r1                   // r1 = 1
sll r1, 63, r1              // r1 = $t^{63}$
mov r1, k                   // k = $t^{63}$
ldx 0, r1                   // r1 = 0
bmulxck r0, r1, r2          // set exc = rex_62 * $t^{62}$ + rex_61 * $t^{61}$ +
                            // + rex_1 * t + rex_0, r2 = old_exc
```

[1121] Note that *bmulxc/bmulxck* can not be used to restore the extended carry register with a value greater than n-1 bits. In particular, *bmulxc/bmulxck* can not be used to restore the extended carry for a subsequent *umulxc/umulxck* instruction, whereas *umulxc/umulxck* can be used to restore the extended carry for a subsequent *bmulxc/bmulxck* instruction.

[1122] Referring now to Fig. 13, a multiplier circuit structure is shown that may be adapted for use in various embodiments of the invention. Fig. 13 shows a multiplier circuit that multiplies the contents X and Y of two n bit registers 1301 and 1303 and outputs a 2n bit result into register R 1305. In the exemplary embodiment shown in Fig. 13, n is 64 bits, thus the result R is 128 bits. The result R is generated as follows. First, each bit of X is multiplied with all bits of Y in partial product generator 1302. The partial product generator outputs n-bit partial products pp63 ... pp0 with $pp_i = x_i * y$. Alternatively, the partial product generator may output the partial products of the Booth encodings of X multiplied by Y. Next, the partial products are summed up in Wallace tree 1304. The Wallace tree 1304 adds partial products and produces a 255-bit intermediate result sum[127..0] and carry [127..1]. The Wallace tree may also generate a 127-bit "XOR multiply" result xor_result [126..0]. Adder circuit 1306 sums up intermediate results sum[127:0] and carry[127..1]. Common implementations of adder circuit 1306 include a ripple-carry-adder, a carry look-ahead adder or a carry-select-adder. Multiplexer 1308 selects the multiplication result r to be either the unsigned integer multiplication r = x * y (xor_multiply=0) or the XOR multiplication r = x ^* y (xor_multiply=1), where ^* indicates XOR multiplication for binary polynomial fields.

[1123] Fig. 14 shows a multiplier circuit, in which the adder circuit is implemented as two 64-bit adders 1404 and 1406. Adder circuit 1404 outputs a carry out bit 1405, which is input into adder circuit 1406. Note that adder circuit 1406 does not generate a carry bit since the product x * y can never be greater than 128 bits.

[1124] Fig. 15 shows a multiply-and-accumulate circuit according to an embodiment of the invention, that multiplies the contents of two 64-bit registers X and Y, adds the contents of a 64-bit extended carry register and outputs a 128-bit result. The upper 64 bits of the result are output into 64-bit extended carry (exc) register 1508 and the lower 64 bits are output into result register 1510. The addition of the 64-bit extended carry (exc) register 1508 is performed in adder circuit 1504. Adder circuit 1504 adds intermediate results sum[63..0], carry[63..1] and exc[63:0] and outputs a 64-bit addition result 1509 and two carry bits 1511 that are input into adder circuit 1506. Multiplexers 1512 and 1514 select between the unsigned integer multiply-accumulate operation (exc, r) = x * y + exc (when multiplexer select xor_multiply=0) and the XOR multiply-accumulate operation (when multiplexer select xor_multiply=1) (exc, r) = x ^* y ^ exc where "^" indicates XOR addition. An exemplary implementation of adder circuit 1504 is shown in Fig. 15a. The adder circuit 1504 includes half adder 1550, full adder 1552 and adder circuit 1554. Note that circuit 1554 calculates {0, sum out [63], ..., sum out [0]} + {carry out [64]...carry out [1], 0}. Common implementations of adder circuit 1554 include a ripple-carry-adder, a carry look-ahead adder or a carry-select-adder.

[1125] Fig. 16 shows a multiply-and-accumulate circuit according to another embodiment of the invention, in which the extended carry is not fully summed up, but stored in a 64+2-bit representation in extended carry registers exc 1608 and exc_cout 1610. Since the carry output of adder circuit 1604 is directly input into register exc_cout 1610, the length of the critical path is shortened allowing for faster implementations. The carry bits in exc_cout are added in adder circuit 1604 (which corresponds to Fig. 15a, where half adder 1550 is replaced with a full adder to add one of the carry bits, and adder circuit 1554 adds the other carry bit to the least significant bit position).

[1126] FIG. 17 shows another embodiment of a multiply-and-accumulate circuit according to an embodiment of the invention, in which the extended carry is stored in a 64+64+2-bit representation in registers exc_sum[63..0], exc_carry[63..0], exc_cout0 and exc_cout1. That representation eliminates adder circuit 1606 in Fig. 16 and allows for a faster implementation of the multiply-accumulate operation. Since in most implementations the lower 64 bits of the 64x64 multiplication will be available earlier than the upper 64 bits, it makes sense to save time on the generation of the extended carry and spend slightly more time on the addition in adder circuit 1702.

[1127] An exemplary implementation of adder circuit 1702 with an array of 4-to-2-compressors 1750 is shown in Fig. 17A.

[1128] FIG. 18 shows another embodiment of a multiply-and-accumulate circuit, in which the extended carry bits in extended carry sum register 1806 and extended carry register 1808 are fed back and added in the Wallace tree 1802. Various embodiments of Wallace trees suitable for use in the present invention are described further herein. The carry out bit in carry out register 1809 supplied from the addition circuit 1810 is also supplied to the Wallace tree 1802.

[1129] FIG. 19 shows another embodiment of a multiply-and-accumulate circuit, in which the extended carry bits in extended carry sum register 1806 and extended carry register 1808 are fed back and added in the Wallace tree 1802. The carry out bit in carry out register 1809 supplied from the adder circuit 1810 is fed back to adder circuit 1810 instead of the Wallace tree 1802. That allows the Wallace tree to begin determining the next sum and carry outputs prior to the addition being completed by adder circuit 1810.

[1130] FIG. 20 shows a multiply-and-accumulate circuit that multiplies the contents of two 64-bit registers X and Y (not shown), adds the contents Z of a third 64-bit register 2001, adds a 64-bit extended carry from register 2005 and generates a 130-bit result (64 bits and two carries from adder 2007 and 64 bits from adder 2009). The addition of the third value Z is performed in adder circuit 2007, which can be implemented similar to Fig. 17a. The circuit shown in Fig. 20 is one embodiment of a circuit to implement the instructions *umulxck* and *bmulxck.*

[1131] In another implementation, shown in Fig. 21A, the value Z from register 2101 is supplied directly into the Wallace tree 2103 along with the feedback of the extended carry sum and carry bits in registers 2105 and 2107. Note that the feedback of the extended carry bits in Fig. 19 and 21A is shown being fed back into the top of the Wallace tree, as described further herein, the feedback is typically into the middle or lower portion of the Wallace tree.

[1132] The Wallace trees (carry save adders) illustrated in the Figures 13- 21 are a collection of full adders arranged so that the partial products of a multiply are reduced (or compressed) to just two terms that are subsequently added in a carry look-ahead adder such as adder 1306 in Fig. 13. A variety of implementations are possible based on the application and the technology used to implement the multiplier. The partial products may be obtained by ANDing all pairs of bits, one from the multiplier and one from the multiplicand (X and Y in Fig. 13). The inputs into the Wallace tree may be outputs of Booth encoding multiplexers.

[1133] Referring to Fig. 21B, a control circuit is illustrated that allows the multiplier circuits described herein, e.g., in Fig. 19, to be utilized for regular multiplications as well as multiplications involving the extended carry register. When the control signal 2110 is zero, the sum or carry feedback to the Wallace tree (or the carry bit into adder 1810) is zero, thus allowing regular multiplications to take place. Also, if the multipliers using the extended carry registers are not on consecutive clocks, the control of zero during the intermediate clocks is needed in order to obtain the correct result. Fig. 21C illustrates another embodiment implementing the *umulxck* and *bmulxck* instructions in which the additions for the feedback carry and sum bits and the extra term are performed in the Wallace tree.

[1134] Before additional details of Wallace trees suitable for utilization in embodiments of the present invention are described, components that are utilized in constructing Wallace trees will be described, which will help provide a basis for understanding some of the issues associated with efficiently designing Wallace trees for various applications described herein.

[1135] Full adders are often combined to create larger units, which can be utilized in Wallace trees. Referring to Figs 22A a full adder is illustrated. The full adder shown has two XOR gates 2201 and a two input multiplexer 2203. A full adder may also use a majority gate instead of the two input multiplexer. Fig. 22B illustrates a 4 to 2 compressor, formed of two full adders. In Figure 22B, one full adder is indicated by the three gates labeled with #1 and the other full adder is labeled with #2. The X output 2207 from one column of the Wallace tree is the Y input 2209 of the next column of the Wallace tree. Fig. 22C illustrates a 5 to 3 compressor. The 5 to 3 compressor has two full adders, but they are combined differently. Figure 23A shows one implementation of an XOR gate, Fig. 23B shows an implementation of a two input multiplexer, and Fig. 23C shows a majority gate made out of NAND gates. Note that the XOR gate and a two input multiplexer illustrated in Figs. 23A and 23B, respectively, have the same latency since the longest path of each circuit is the same. The majority gate illustrated in Fig. 23C also takes about the same amount of time since the increased latency of a three input NAND gate approximately offsets the absence of the inverter.

[1136] The XOR multiplier (mul) majority gate illustrated in Fig. 23D is only a little slower than the simple majority gate shown in Fig. 23C and is useful is applications performing both integer and XOR multiplication. When the mul select input 2320 is one, it is a majority gate. However, when it is zero, the output is always zero. When performing an XOR multiplication, the mul select input is set to zero forcing the output of the majority gate to zero, which forces carry outputs utilized in forming the XOR result to zero. Referring to Fig. 22C, that forces the output from XOR gate 2211 to depend only on the result of the XOR of inputs D and E produced by XOR gate 2212. The output 2214 from the multiplexer 2215 is assumed to not be utilized in forming the XOR multiplication result.

[1137] For the 4 to 2 compressor shown in Fig. 22B, use of the mul majority gate causes the output 2207 to be forced to zero for XOR multiplication operations and the Y input from a previous mul majority gate is also zero. The CO output on node 2208 is assumed not used in forming the XOR multiplication result.

[1138] Note that the full adder in Fig. 22A and the 5 to 3 compressor illustrated in Fig. 22C each takes two levels of logic whereas the 4 to 2 compressor shown in Fig. 22B takes 3 levels of logic.

[1139] The efficiency is a measure of what percentage of bits are eliminated for each level of logic. The full adder gets rid of 33.3% of its input bits (3 input bits, 2 output bits) in two levels of logic, so its efficiency is one minus the square root of 66.6%, or 18.4%. The 4 to 2 compressor is more efficient since it gets rid of 50% of its input bits in 3 levels of logic. This is one minus the cube root of 50%, or 20.6%. The 5 to 3 compressor is even more efficient. It gets rid of 40% of its input bits in two levels of logic, and one minus the square root of 60% is 22.5%.

[1140] Referring now to Fig. 24 an example of a Wallace tree column for a 64x64 multiplier with Booth encoding is shown. The Wallace tree columns illustrated in Figs. 24 -32 are examples for the longest column of a 64 by 64 multiplication where Booth encoding is used. This results in 33 terms to be reduced to two terms in the Wallace tree. Two notations are used in the Figs. 24-32 that require explanation. An X on a wire indicates that the wire before the X goes to the corresponding position in the next column and the wire after the X came from the corresponding position in the previous column. A second X (see, e.g., Fig. 25) indicates that the wire before the two X's goes to the corresponding position two columns over and the wire after the X's comes from the corresponding position two columns previous. A diamond on a wire indicates that the wire before the diamond goes to the corresponding position in the previous column and the wire after the diamond came from the corresponding position in the next column.

[1141] Fig. 24 shows an exemplary Wallace tree structure. The 32 multiplexers 240 provide a Booth encoding multiplexer. As Booth encoding is known in the art, it will not be discussed further herein. One input supplied on node 241 does not need the Booth encoding multiplexer, as its output may be obtained from a simple AND gate (the AND gate is not shown). Each of the first four rows of logic 242, 243, 244, 245 has 4 to 2 compressors, and so uses 3 levels of logic for each row for a total of 12 levels of logic. The final row has only a full adder 246. The full adder 246 takes another 2 levels of logic, so the total number of levels of logic is 14. In Figure 25, the last 4 to 2 compressor 245 (Fig. 24) is replaced with a 5 to 3 compressor 251. That reduces the total number of levels of logic to 13. Figure 26 uses 5 to 3 compressors extensively and so only 12 levels of logic are needed.

[1142] Now that several variations of Wallace trees have been shown, exemplary Wallace trees are shown that provide for the extended carry feedback required by the *umulxc* and *umulxck* (and *bmulxc* and *bmulxck*) instructions in the embodiments illustrated, e.g., in association with Figs. 19 and 21.

[1143] In order to accommodate both the XOR multiply-accumulate function and the integer multiply-accumulate for the *bmulxc* and *umulxc* instructions, three feedback terms may be utilized as illustrated in the exemplary embodiment shown in Fig. 27. One feedback term supplied on node 271 is for the XOR multiply when used in the multiply-accumulate mode, and the two terms (sum and carry) are supplied on nodes 272 and 273 for the integer multiply when used in the multiply-accumulate mode. The XOR result is supplied on node 274 and the sum and carry multiplication result from this column is supplied on nodes 275 and 276. Thus, for each Wallace tree column, at most one carry bit and one sum bit of the extended carry value from the high portion of the previous multiplication is fed back. For the XOR multiply, at most one bit is fed back.

[1144] Note that the number of levels of logic from the feedback inputs to the outputs is very small. For the XOR result, it is two levels from input 271 to output 274, and for the integer multiply terms it is 6 levels from inputs 272, 273 to the outputs 275, 276. This can be done in less than one clock, giving time for logic to decide whether (and when) feedback should be used. Note that the first row of the Wallace tree has only full adders. That is because the XOR result of all the inputs is needed in the XOR multiply mode, and the regular majority gate used in the compressors interferes with obtaining the XOR result. The Wallace tree column illustrated in Fig. 27 has 14 levels of logic.

[1145] Referring now to Fig. 28 shown is an exemplary embodiment of a Wallace tree column used in a 64x64 multiplier with Booth encoding that supports the *umulxc* and *bmulxc* instructions for both integer multiply-accumulate and XOR multiply-accumulate. The embodiment in Fig. 28 utilizes the mul majority circuit 234 shown in Fig. 23D in the 5:3 compressors 280 shown in Fig. 22C. The use of the mul majority circuit allows the first row to not be restricted to just full adders and still support the XOR multiply function by setting the mul input to the 5:3 compressor to 0, forcing the output of the majority circuit in the 5:3 compressor to 0 for XOR multiply operations. The use of the mul majority circuit results in 13 levels of logic. Note that the XOR result 282 comes from the output of the XOR gate 2210 (Fig. 22C) whose inputs are the A and B inputs. The extended carry sum and carry feedback bits are supplied on nodes 284 and 285.

[1146] Referring now to Fig. 29 shown is an exemplary embodiment of a Wallace tree column is shown used in a 64x64 multiplier with Booth encoding that supports the *umulxc* and *bmulxc* instructions for both integer multiply-accumulate and XOR multiply-accumulate. The embodiment in Fig. 29 utilizes the mul majority circuit 234 shown in Fig. 23D in the 4:2 compressors 2902 shown in Fig. 22B. The use of the mul majority circuit in the 4:2 compressors of Fig. 29 results in 14 levels of logic. A series of multiplexer 2900 are utilized to provide the Booth encoding. Outputs of the multiplexer are supplied to the 4:2 compressors 2902, whose outputs are supplied to the full adders 2905 and the 5:3 compressor 2907. The series of full adders 2909, 2910, 2911, and 5:3 compressor 2913 are also utilized in the exemplary tree structure. Note that the full adder 2910 receives a first extended carry input on node 2915 and full adder 2911 receives a second extended carry input on node 2917. For the 64 x 64 multiplier, the outputs of the Wallace tree columns generating the high order bits that represent the extended carry are fed back into the Wallace tree columns generating the low order bits. The two outputs from the 4:2 compressor 2919 are those bits supplied to the carry look-ahead adder. Referring to Fig. 12, those bits would be, e.g., bits S3 and C3.

[1147] Note that Fig. 29 also includes an XOR gate 2921. That XOR gate receives the result of the XOR of the 32 inputs of the Wallace tree and is also supplied with the feedback of a high order bit of the XOR multiplication result, which is fed back to be XORed with the XOR result generated by the Wallace tree, thus adding a bit of the extended carry into a column of the Wallace tree for the XOR multiplication result. The carry look-ahead adder is not needed to implement the XOR multiplication utilized in binary polynomial field operations since no carry terms are generated. While the XOR multiplier may be integrated into the multiplier that performs integer multiplication, other embodiments contemplate the XOR multiplier and integer multipliers being separate. Further, some embodiments may utilize only XOR or only integer multipliers. Note also, while Booth encoding is illustrated, other embodiments may not utilize Booth encoding.

[1148] The Wallace tree column, such as the column illustrated in Fig. 29 can take two or more clock cycles to generate the result supplied to the carry look-ahead adder and the carry look-ahead adder can take nearly a clock cycle. If the carry output of the carry look-ahead adder (bit CC4 in Figs. 11-12), were also fed back into the Wallace tree, then there would have to be an approximately one clock cycle delay between chained multiplies, which could significantly reduce performance. While that may be acceptable in certain embodiments, where for example, the operation is performed relatively infrequently so performance is not an important criterion, it is generally preferable to obtain more efficient performance if available. Thus, by feeding back the carry output of the carry look-ahead adder, into the carry look-ahead adder, as illustrated in, e.g., Figs. 19 and 21, more efficiency is gained.

[1149] For the *umulxck* and *bmulxck* instructions, where an additional term needs to be added in, Figure 27 can be modified into Figure 30. The half adder 277 in Fig. 27 is changed to a full adder 3001 in Fig. 30 to accommodate the extra term supplied on node 3003 from register Z (Fig. 20). The XOR multiply considerations are preserved. Thus, each Wallace tree column for the low order bits gets both feedback bits from the extended carry register and the bit to be added from the source operand rs2 (Z in Fig. 20) specified in the *umulxck* instruction (*umulxck rs1, rs2, rd*) and the *bmulxck* instruction.

[1150] Fig. 28 can also be modified for the *umulxck* and *bmulxck* instructions as shown in Fig. 31. The extra term from the source operand rs2 specified in the *umulxck* instruction (*umulxck rs1, rs2, rd*) and the *bmulxck* instruction is input into the Wallace tree on node 3103 to full adder 3110 where the XOR feedback was in Fig. 28. The XOR feedback supplied on node 3111 is input into XOR gate 3112.

[1151] Fig. 29 can also be modified to support the *umulxck* and *bmulxck* instructions as illustrated in Fig. 32. To do this, three of the full adders in the second row are replaced with two 5 to 3 compressors. Note that the 33rd Booth encoding input on node 3201 that, in Figure 29 (node 292) contributed to the XOR output, does not contribute to the XOR output. This is acceptable because when obtaining the XOR result, the 33rd Booth encoding input is always zero. Thus it doesn't matter whether the value is included in the XOR result.

[1152] The instructions proposed herein provide significant performance advantages. The instructions defined herein compute the product once and save the upper 64-bit result for the next operation. The new instruction can propagate the upper 64 bits of a product into a subsequent operation without incurring delay. That helps reduce the delay and the number of registers needed to store intermediate results. The extended carry register saves the upper-64-bit result and accumulates it into the next operation.

[1153] The embodiments described above are presented as examples and are subject to other variations in structure and implementation within the capabilities of one reasonably skilled in the art. The details provided above should be interpreted as illustrative and not as limiting. Variations and modifications of the embodiments disclosed herein, may be made based on the description set forth herein, without departing from the scope of the invention as set forth in the following claims.

Appendix A

umulxck rs1, rs2, rd : rd <- lower 64-bits of (rs1*K + rs2 + exc)
exc <- higher 64-bits of (rs1*K + rs2 + exc)

```
1) ldx [a0], %13; xor %g0, %g0
2) ldx [b0], %l4; mov %13, %K
3) ldx [b1], %l5; umulxck %g0, %g0, %g1 ! exc=0
4) ldx [b2], %l6; umulxck %l4, %g0, %i0
5) ldx [b3], %l7; umulxck %l5, %g0, %i1
6) umulxck %l6, %g0, %i2
7) umulxck %l7, %g0, %i3
8) umulxck %g0, %g0, %i4
9) ldx [a1], %13
10) mov %13, %K
11) umulxck %l4, %i1, %i1
12) umulxck %l5, %i2, %i2; stx %i0, [dest]
13) umulxck %l6, %i3, %i3
14) umulxck %l7, %i4, %i4
15) umulxck %g0, %g0, %i5
16) ldx [a2], %13
17) mov %13, %K
18) umulxck %l4, %i2, %i2
19) umulxck %l5, %i3, %i3; stx %i1, [dest+8]
20) umulxck %l6, %i4, %i4
21) umulxck %l7, %i5, %i5
22) umulxck %g0, %g0, %i6
23) ldx [a3], %13
24) mov %13, %K
25) umulxck %l4, %i3, %i3
26) umulxck %l5, %i4, %i4; stx %i2, [dest+16]
27) umulxck %l6, %i5, %i5
28) umulxck %l7, %i6, %i6
29) umulxck %g0, %g0, %i7
30)
31)
32)
33) stx %i3, [dest+24]
34) stx %i4, [dest+32]
35) stx %i5, [dest+40]
36) stx %i6, [dest+48]
37) stx %i7, [dest+52]
```